Programmation objets, web et mobiles (JAVA)

# Cours 2 - Héritage

Licence 3 Professionnelle - Multimédia
**Philippe Esling** (esling@ircam.fr)
Maître de conférences – UPMC
Equipe représentations musicales (IRCAM, Paris)


# <Gift retour> Bataille de cartes

Faire un programme :

1. Créer une classe *Bataille* qui contiendra le **main**
2. Créer une classe *Carte*
    1. Contient un constructeur pour créer une carte
    2. Attributs : *couleur* (parmi un tableau statique) et *valeur* (idem)
    3. Méthode : accesseurs, modificateurs et comparateur
3. Créer une classe *Joueur*
    1. Propriétés : tableau de cartes et compteur de points
    2. Méthode : tire une carte et ajoute une carte
4. Ecrire le programme de jeu principal
    1. Crée deux joueurs.
    2. Initialise un paquet de cartes et le mélange aléatoirement (**Math.random**)
    3. Effectue une boucle de jeu en affichant les scores (**System.out.println**)
    4. Affiche le vainqueur de la partie

**45 minutes - Noté**

```java
package pobj.cours3;

public class Point {
  // attributs
  private double x, y;
  // constructeurs
  public Point(double a, double b){x=a;y=b;}
//  public Point(){x=0;y=0;}
  // accesseurs
  public double getX(){return x;}
  public double getY(){return y;}
  // me'thodes
  private void moveto (double a, double b){x=a;y=b;}
  public void rmoveto (double dx, double dy){x+=dx;y+=dy;}
  public double distance(){
    double x = this.getX();
    double y = this.getY();
    return Math.sqrt(x*x+y*y);
  }
  // me'thodes pre'de'finies (standards)
  public String toString(){return ("("+x+","+y+")");}
}
```

# Lecture d'une classe

- Quel est le nom du paquetage de la classe ?
- Quel est le nom de la classe ? nom court ou nom qualifié par le paquetage ?
- Quel est le nom du fichier ? Quel est le chemin d'accès ?
- Quels sont les constructeurs ? leurs signatures ?
- Quelles sont les méthodes publiques ? leur signature ?
- Quelles sont les méthodes privées ? leur signature ?
- Quelles sont les méthodes prédéfinies ?

# Package

Regroupement de classes et d'interfaces dans un archivage hiérarchique.

```
1  package nom[.nom]* ;
```

La déclaration d'un paquetage est la première instruction d'un programme.

```
1  import nomqualifie;
2         nomqualifie.*;
```

- autorise les références abrégées.
  collisions interdites.
- Sans rien préciser dans son programme, celui-ci est considéré faisant partie du "paquetage anonyme" par défaut.
- La hiérarchie des paquetages suit la hiérarchie des catalogues du système.
- L'appartenance à un paquetage modifie les règles de visibilité.

# Méthodes prédéfinies

- égalité et copie

```
1 boolean equals(Object obj);
2 protected Object clone();
```

- finalisation

```
1 protected void finalize()
```

- introspection

```
1 Class getClass();
```

- threads

```
1 void notify()
2 void notifyAll()
3 void wait()
```

- valeur de hachage pour l'objet

```
1 int hashCode()
```

- conversion en chaînes de caractères

```
1 String toString()
```

# Java statique : programmation modulaire

Les variables et méthodes de classes :

- sont déclarées avec **static**
- existent dès le chargement de la classe (sans instance)
- existent en un unique exemplaire
- accessibles par la notation “point”

Une classe qui ne contient que des champs statiques peut être considérée comme un module classique (Ada (LI330), OCaml (LI332), ...) :

où les méthodes sont les fonctions du modules et les variables les variables globales.

```
1  public static double sqrt(double a)  // java.lang.Math
2  static void gc()                     // java.lang.System
3  static PrintStream out               // java.lang.System
```

# Java dynamique : programmation objet

Les variables et méthodes d'instances :

- ne sont pas déclarées avec **static**
- sont allouées à la création d'une instance (**new**)
  - état local à l'instance pour les variables
  - table des méthodes commune pour toutes les instances de la même classe
- existent pour chaque instance
- accessible aussi par la notation "point" si **public**

o.m(a) :appel de méthode (envoi du message) `m` avec le paramètre a sur l'objet o d'une classe c de construction (c'est-à-dire un objet o créé par un constructeur défini dans la classe c).

# Représentation des objets (1)

Un objet est une instance d'une classe ! ! !

Une classe peut avoir plusieurs instances.

Pour cela un objet est formé :

- d'un état local contenant les valeurs des variables d'instance
- + la valeur de lui-même (**this**)
- et d'une table des méthodes d'instance contenant l'ensemble des méthodes d'instance de la classe et les méthodes prédéfinies (voir le cours 3 sur l'héritage).

# Représentation des objets (2)

Pour cela on peut représenter un objet comme un enregistrement contenant les variables d'instances (+ **this**) et les méthodes d'instances.

*Pour en savoir plus* : accessible en ligne

> *Implementing statically typed object-oriented programming languages. Roland Ducournau ACM Computing Surveys, 2010*

# Constructeur

Chaque classe possède au moins un constructeur pour initialiser un nouvel objet de ce type.

- méthode de même nom que la classe
- sans type de retour
- appelé dès l'allocation (**new**)

L'allocation par un constructeur est **explicite** en utilisant la construction **new.**

```
1  Point p0 = new Point();
2  Point p1 = new Point(2,3);;
3  IStack s1 = new StackAL();
4  IStack s2 = new StackA(20);
```

# Emploi de this

- **this** correspondant à une référence sur l'objet en cours d'exécution d'une méthode, permet de le référencer :

```
1  public double distance(Point p2){
2    double dx = p2.getX() - this.getX();
3    double dy = p2.getY() - this.getY();
4    return Math.sqrt(dx*dx + dy*dy);
5  }
```

- peut être utilisé en notation qualifiée

```
1  Point (double x, double y) {this.x = x; this.y = y;}
```

- ne peux pas être utilisé dans une méthode statique car il faut qu'un objet **this** ait été créé.

```
1  pobj/cours2/Bad.java:5: non-static variable this cannot be referenced ↩
       from a static context
2    public static void m(){System.out.println(this.x);}
3                                                ^
4  1 error
```

# Mémoire et Egalité (1)

- Du point de vue mémoire un objet est une référence sur une zone allouée contenant les variables d'instance et la table des méthodes d'instance ;
- la valeur **null**, le pointeur nul, est une valeur acceptable pour toute variable dont le type n'est pas primitif ;
- c'est la valeur par défaut de toute variable du type d'une classe. pointeur = référence = adresse mémoire
  pointeur nul = **null**.

```
1 Point p0 = new Point(2,3);
2 Point p2 = null;
```

# Mémoire et Egalité (2)

- les objets et les tableaux sont des références ! ! !
  pas les types de base
- opérateur == teste uniquement les adresses, pas le contenu de la zone pointée
- Utiliser alors la méthode **equals(o)** prédéfinie (héritée d'*Object*) et que l'on peut redéfinir (to override) et/ou surcharger (to overload, slide 34).
- Par défaut **equals** teste l'égalité physique.

```
1  public boolean equals (Point p) {  // surcharge
2    if (this == p) return true;
3    else
4      if (p == null) return false;
5      else
6        return ( (this.getX() == p.getX()) &&
7                 (this.getY() == p.getY()) );
8  }
9
10 public boolean equals (Object o) { // rede'finition
11   return equals ((Point)o);
12 }
```

# Mémoire et Egalité (3)

- Allocation explicite (**new**)
- mais récupération automatique (GC).
  - GC : Garbagge Collector (Glaneur de Cellules)
  - description des techniques du GC Java (Oracle)
    - générationnel, compactant, concurrent, parallèle
  - peut être déclenché explicitement

```
1 System.gc()
```

  - et paramétré au lancement de la JVM
  - possibilité de redéfinir une méthode `finalize` qui sera exécutée avant la libération d'un objet par le GC

$\Rightarrow$ pour en savoir plus sur les algorithmes de GC voir cours 5 (LI332).

# Langage de modélisation

(très fortement inspiré du cours de Yann Régis-Gianas (P7))

langage UML de modélisation graphique (Unified Modeling Language)

- Notation semi-formelle standardisée de modélisation développée par l'OMG (Object Management Group)
- Utilisation généralisée pour rédiger des documents de travail.
- Accent mis sur la description, pas sur la justification.
- On segmente la représentation du système en vues.
- Différents types de vue : Dynamiques, statiques

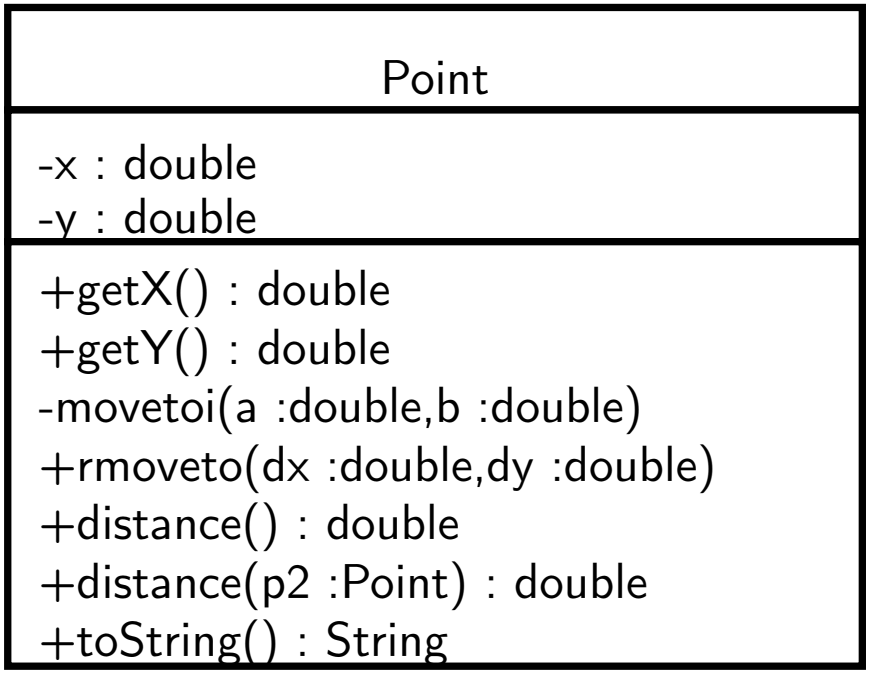
$\Rightarrow$ voir cours « Introduction à la modélisation des systèmes logiciels avec UML » (LI342)

# Diagramme de classes

- un diagramme de classe est une vue statique décrivant l'organisation des composants.
- Un composant peut être :
  - **une classe** ;
  - une classe dans un état donné ;
  - un objet ;
  - un nœud ou une ressource logicielle ;
  - un rôle ;
  - **une interface** ;
  - un acteur ;
  - un cas d'utilisation ;
  - un sous-système ;
- Deux composants peuvent être en relation :
  - de généralisation ;
  - **d'association** ;
  - **de dépendance** ;
  - de réalisation ;
- une notation existe pour chacun de ces types de composant et de relation

# Composant : classe

- Premier bloc : nom de la classe (en *italique* si abstrait)
- Deuxième bloc : liste des attributs, chaque ligne contient
  - un modifieur : + pour public, - pour privé, # pour protégé
  - le nom de l'attribut et le type de l'attribut
- Troisième bloc : liste des opérations, chaque ligne contient :
  - un modifieur : + pour public, - pour privé, # pour protégé
  - le nom de l'opération, ses arguments et leur type, et le type de retour



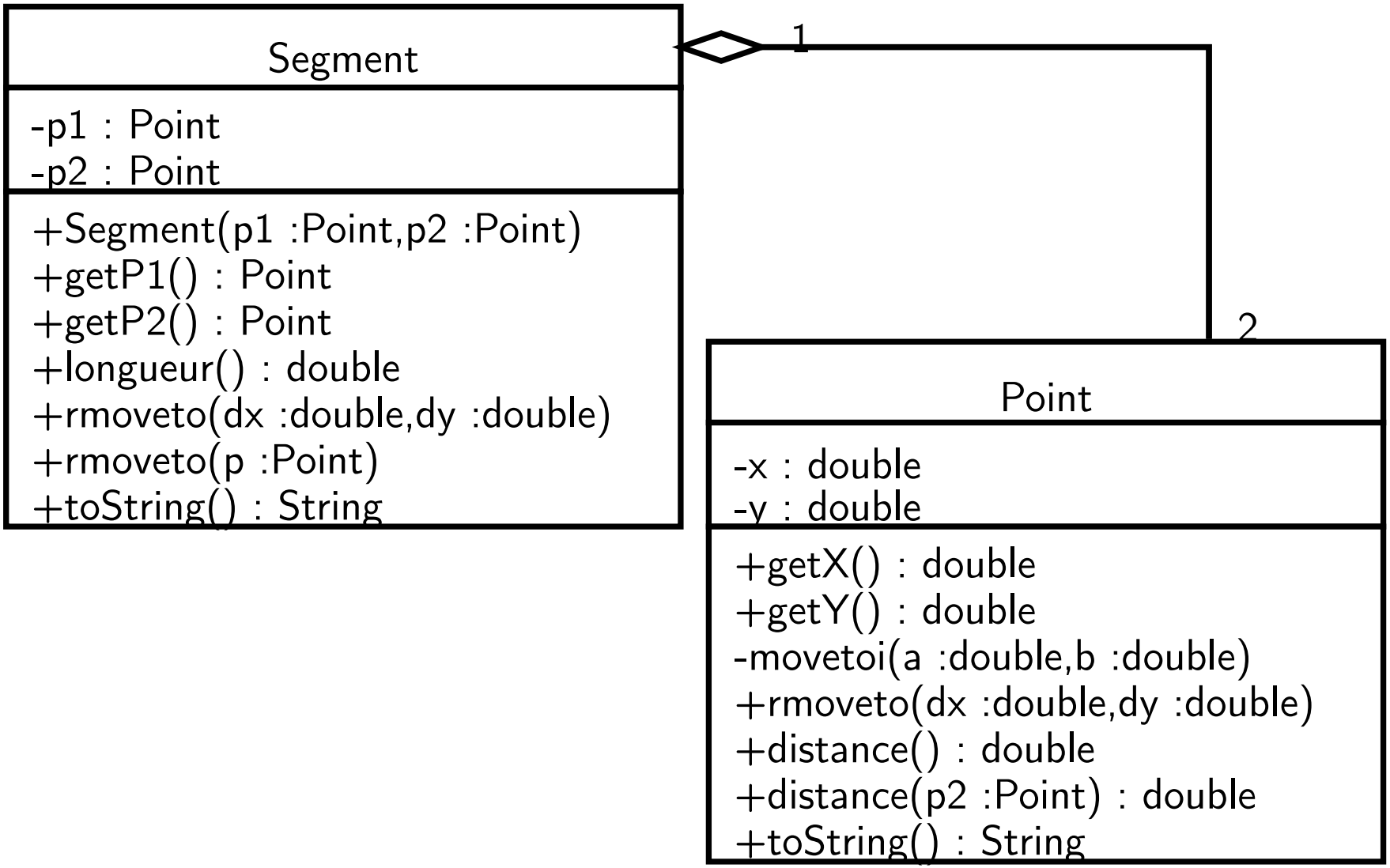
# Relation : dépendance

- une relation de dépendance est notée par une ligne en pointillé terminée par une flèche
- Elle dénote l'existance nécessaire de certains composants pour le bon fonctionnement d'un composant particulier
- Il y a différents types de dépendances : `call`, `bind`, `access`, `derive`, `friend`, `import`, `instantiate`, `parameter`, `realize`, `refine`, `send`, `trace`, `use`.

# Relation : association (1)

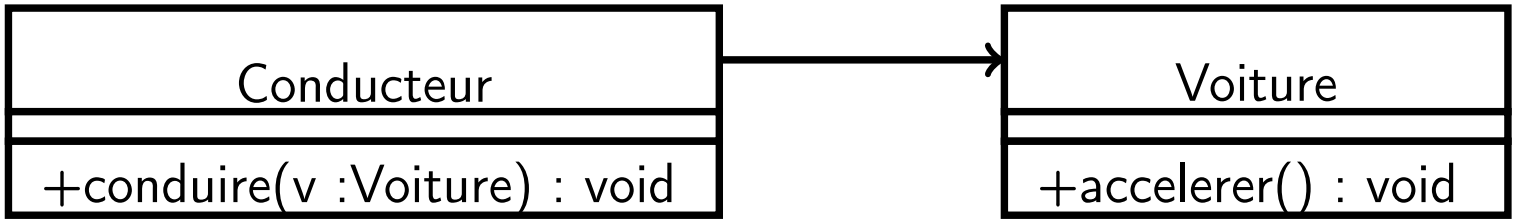
- Une relation d'association est notée par une ligne pleine dont les extrêmités sont annotées (optionnellement) par :
  - une multiplicité :
    - * signifie « un nombre indéterminé »
    - $n \in N$, un nombre $n$ fixé
    - $m...n$, $(m, n) \in N^2$, un nombre entre $m$ et $n$.
  - un losange plein signifie que l'association est une composition.
  - un losange vide signifie que l'association est une agrégation.
- La composition donne le droit de vie ou de mort à un objet sur un autre (pas de partage).
- l'aggrégation (ou composition) : relation « **A-UN** » entre deux classes (« A-DEUX », « A-DES », « EST-COMPOSE-DE », etc)
- Exemples :
  - Voiture «**A-UN**» Moteur
  - Segment «**A-DEUX**» Point

# Relation : association (agrégation) (2)



# Relation : association (3)

- un Conducteur « conduit » une Voiture
- sens de l'association « conduit »

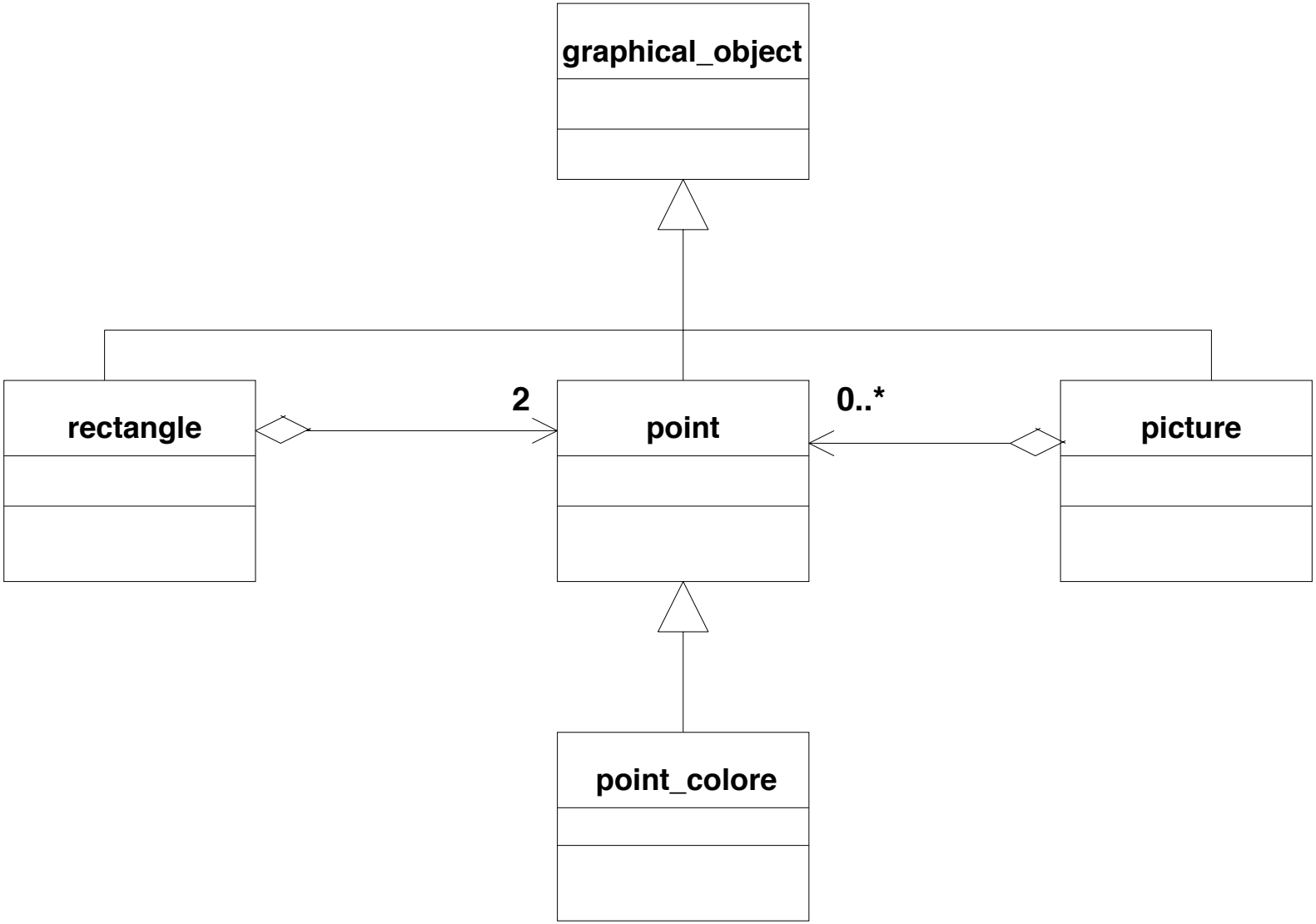


# Relation de généralisation (1)

- Une relation de généralisation est notée par une ligne pleine et termine par une flèche.
- entre classes, elle dénote qu'une classe est plus générale qu'une autre ;
- entre deux classes A et B, elle exprime que « B est un A», c'est-à-dire qu'un objet de la classe B peut être utilisé en place et lieu d'un objet de la classe A

  $\Rightarrow$ principe de subsomption (déjà rencontré quand une classe implante une interface).

# Relation de généralisation (2)



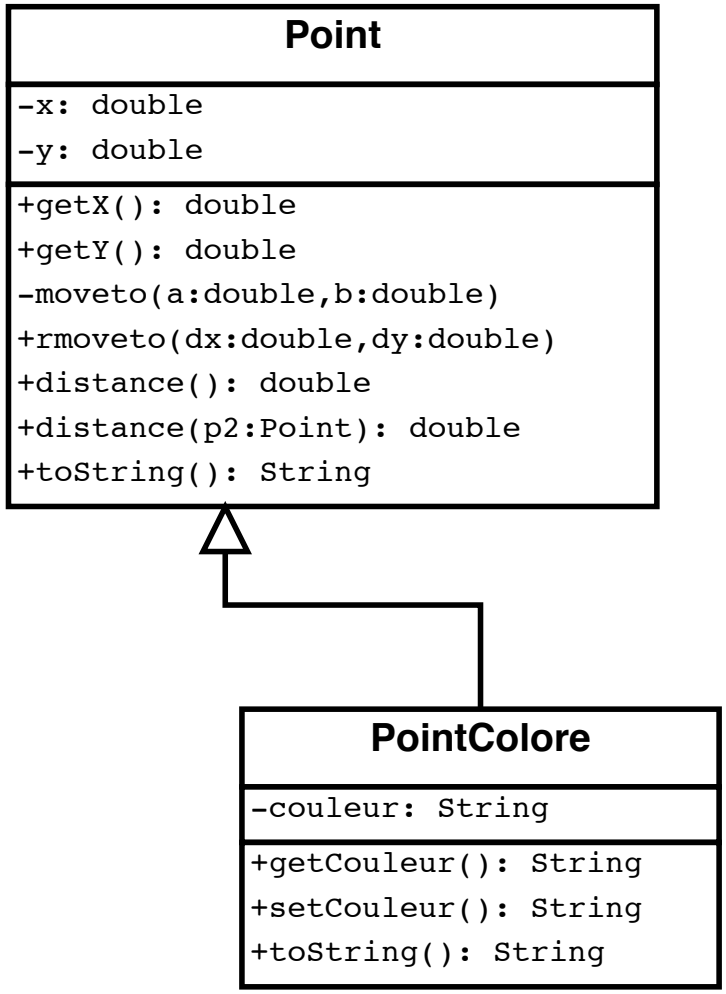
# Héritage

- Concept primordial de la programmation objet :
- Extension du comportement d'une classe existante tout en continuant à utiliser les variables et les méthodes décrites par la classe originale.
- En Java toute définition de classe étend une classe existante.
- En Java l'héritage met en relation de généralisation la sous-classe à la super-classe.
- Si l'on ne précise rien on étend la classe **Object**

  $\Rightarrow$ une classe hérite toujours d'une autre ! ! !

# Héritage : relations

Classes `Point` et `PointColore` :



# Héritage : classe Point

```java
package pobj.cours3;

public class Point {
  // attributs
  private double x, y;
  // constructeurs
  public Point(double a, double b){x=a;y=b;}
//  public Point(){x=0;y=0;}
  // accesseurs
  public double getX(){return x;}
  public double getY(){return y;}
  // me'thodes
  private void moveto (double a, double b){x=a;y=b;}
  public void rmoveto (double dx, double dy){x+=dx;y+=dy;}
  public double distance(){
    double x = this.getX();
    double y = this.getY();
    return Math.sqrt(x*x+y*y);
  }
  // me'thodes pre'de'finies (standards)
  public String toString(){return ("("+x+","+y+")");}
}
```

# Héritage : classe PointColore

```java
package pobj.cours3;

public class PointColore extends Point {
  private String couleur;
  public PointColore(double x, double y, String c) {
      super(x,y); this.couleur = c;
  }
  public PointColore() { couleur = "INDEFINIE";}
  public String getCouleur(){return couleur;}
  public void setCouleur(String c){couleur=c;}
  public String toString(){
    return super.toString() + "-" + this.getCouleur();
  }
}
```

# Héritage : exécution

```java
package pobj.cours3;

class ExPoints {
  public static void main(String[] args) {
    Point p0 = new Point();
    Point p1 = new Point(2,3);
    PointColore pc0 = new PointColore();
    PointColore pc1 = new PointColore(2,3,"Bleu");

    System.out.println(p0 + " " + p1);
    System.out.println(pc0 + " " + pc1);
  }
}
```

```
> java pobj/cours3/ExPoints
(0.0,0.0) (2.0,3.0)
(0.0,0.0)-INDEFINIE (2.0,3.0)-Bleu
```

# super et this (1)

```java
1  public String toString(){
2    return super.toString() + "-" + this.getCouleur();
3  }
```

- **this** : représente l'objet courant (celui qui effectue le calcul) de la classe de définition
- **super** : représente l'objet courant vu de la classe ancêtre

$\Rightarrow$ permet de distinguer les redéfinitions (`super.toString()`) ou de nommer les constructeurs ancêtres (`super(x,y)`).

# super et this (2)

La liaison avec **super** peut être résolue à la compilation (on connaît l'adresse de la méthode de la classe ancêtre). Cela ne marche qu'à un niveau (il n'y a pas de `super.super.m()`).

```
1  public String toString(){
2    return super.toString() + "-" + this.getCouleur();
3  }
```

Ce sera toujours le `toString` de `Point` qui sera appelé dans la méthode `toString` de `PointColore`.

S'il n'y a pas d'appel explicite d'un constructeur de **super**, alors l'appel `super();` est ajouté en 1ère instruction du constructeur de la classe fille. On aurait pu écrire :

```
1  PointColore(){couleur="INDEFINIE";}
```

à la place de

```
1  PointColore(){super(); couleur="INDEFINIE";}
```

nécessite un constructeur sans paramètre dans la classe ancêtre.

# Héritage et liaison tardive

Soit la méthode `distance` de la classe `Point` suivante :

```
1  public double distance() {
2   return  Math.sqrt(
3       this.getX() * this.getX() + this.getY() * this.getY()
4     ) ;
5  }
```

si on redéfinit la méthode `getX` de la classe `PointColore` :

```
1    double getX(){return 2 * x;}
```

Alors l'expression :

```
1  (new PointColore(2,3,"Bleu")).distance()
```

retourne la valeur 5.0 $(\sqrt{16+9}) \neq \sqrt{4+9}$

$\Rightarrow$ permettant de modifier le comportement de méthodes héritées.

le comportement de `distance` est modifié sans changer le corps de la méthode, seulement en modifiant `getX`.

# Mini-projet : Lecteur de fichiers

- Fichier = système de communication/spécification pour les projets collaboratifs
- Simplification de l'architecture des IHM multi-composants, réduction des dépendances

3 manières de gérer les fichiers. En JAVA, une logique de flux

- Approche générale, *à l'ancienne*
  - (1) Lecture/écriture ASCII
- Approche Objet
  - (2) Serialization
  - (3) Externalization

# Mini-projet : Lecteur de fichiers

1. **Fichiers** : lire les noms, vérifier l'existence, vérifier la possibilité d'écriture...
   - Equivalent des fonctions `dir, cd, ...` Mais à l'intérieur de JAVA
2. Une fois le fichier ciblé, **l'ouvrir** et **lire** ce qu'il y a dedans
3. **Créer** un fichier et/ou **écrire** dedans
4. ... D'autres choses se gèrent comme les fichiers
   - Clavier, Réseau...

# Mini-projet : Lecteur de fichiers

**Classe File**

Cette classe permet de gérer les fichiers :

- test d'existance
- distinction fichier/répertoire
- copie/effacement
- ...

- boolean canExecute()
- boolean canRead()
- boolean canWrite()
- boolean delete()
- boolean isDirectory()
- boolean isFile()
- File[] listFiles()
- boolean mkdir()

Nombreuses opérations très intéressantes concernant la manipulation des fichiers

# Mini-projet : Lecteur de fichiers

1. `File` : désigner un fichier
2. `FileInputStream` : création de cet objet = ouverture en lecture du fichier
   - Des **exceptions** à gérer
   - Penser à **fermer** les fichiers ouverts

```
1  FileInputStream in = null;
2  File f = new File("xanadu.txt");
3   try {
4       in = new FileInputStream(f); // ouverture du fichier
5       // Throws: FileNotFoundException : => try/catch
6
7       // OPERATIONS DE LECTURE
8
9       }
10  } finally {
11      if (in != null) {
12            in.close();
13      }
14  }
```

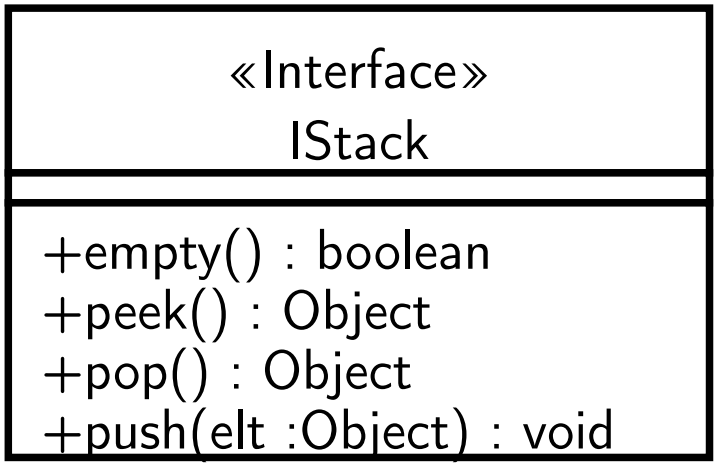
Faire un programme :

1. Classe LecteurFichier: Lis le nom d'un fichier (donné au constructeur), l'affiche sur la console
2. Classe LecteurReverse: Ne modifie que la fonction affiche pour afficher à l'envers

**30 minutes - Noté**

# Composant : interface

- une interface représente un ensemble d'opérations caractérisant un comportement




- version simplifiée représentée par un nom et un cercle

# Interfaces

```
1  [ public ] interface l_intf [ extends [ a_intf]+ ] {
2    declarations
3  }
4
5
6  [public] class la_classe [extends superclasse]
7     [implements [l_intf]+] {
8  }
```

- une interface est constituée de déclaration de variables et d'entêtes de méthodes
- elles sont non instanciables
- seule une classe peut implanter une ou plusieurs interfaces, c'est-à-dire répondre à la spécification demandée

# Héritage et interface (1)

Une classe qui étend une classe qui implante une interface l'implante aussi.

exemple : interface `Copiable`

```
1  interface Copiable {
2    Object copier()
3  }
```

```
1  class Point implements Copiable {
2    ...
3    public Object copier() {
4      return new Point(this.x,this.y);}
5  }
6  class PointColore extends Point {
7  ...
8    public Object copier() {
9      return new PointColore(this.x,this.y, this.couleur);}
10 }
```

```
1  Copiable cp1 = new Point(10,20);
2  Copiable cp2 = new PointColore(1,1,"Rouge");
```

# Héritage et interface (2)

Une interface peut hériter d'une autre interface et même de plusieurs interfaces. Les méthodes de cette interface correspondent à l'union des méthodes héritées et des méthodes déclarées.

Dans le paquetage `javax.swing.event` on a :

```
1 public interface MouseInputListener
2 extends MouseListener, MouseMotionListener
```

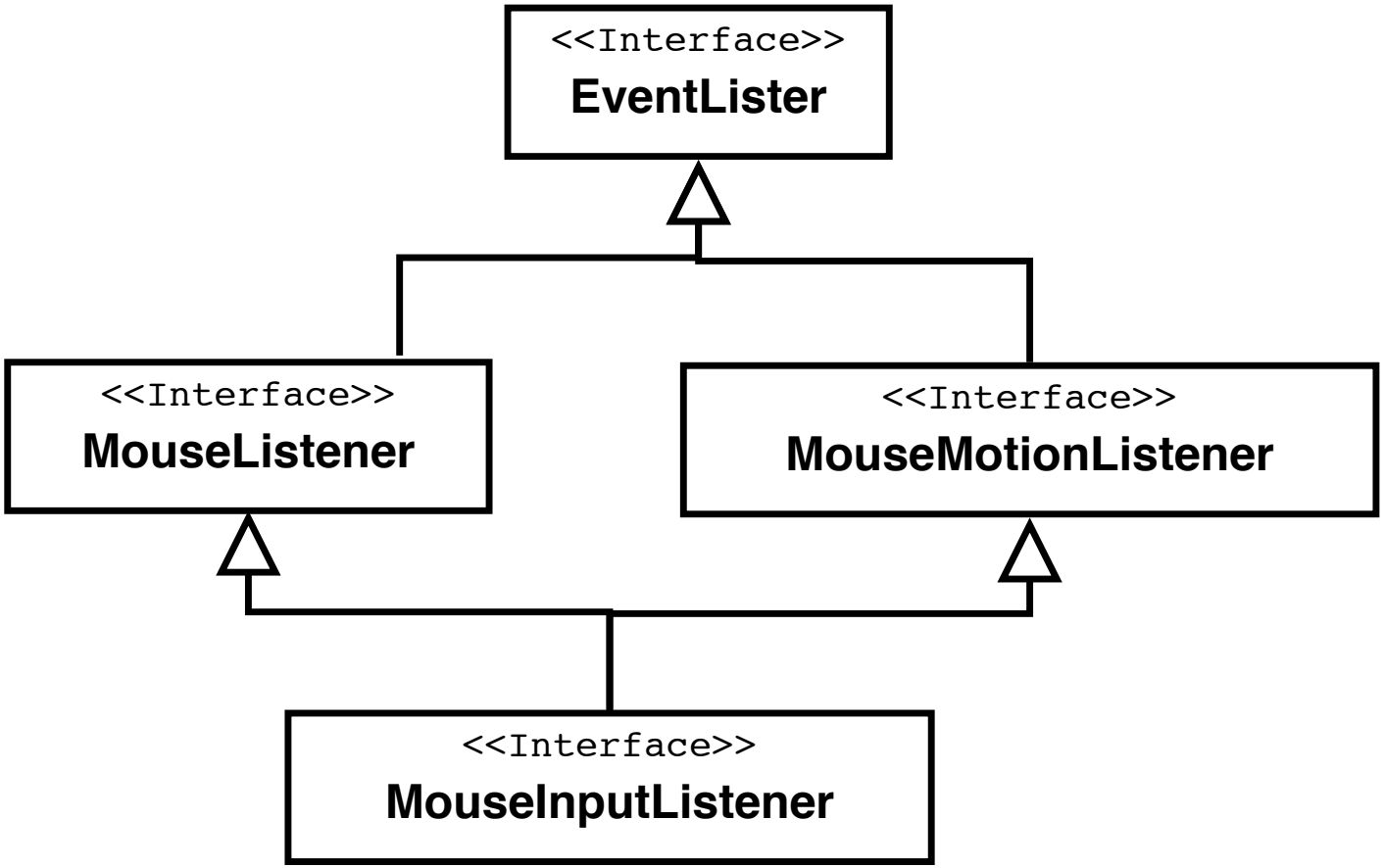
sachant que `MouseListener` et `MouseMotionLIstener` étendent `EventListener` :

```
1 public interface MouseListener
2 extends EventListener;
3
4 public interface MouseMotionListener
5 extends EventListener;
```

Au final l'interface `MouseInputListener` a trois super-interfaces.

# Héritage et interface (3)

Représentation graphique de l'exemple précédent



# Surcharge et rédéfinition

- redéfinition ou surcharge
  - même signature : redéfinition
  - surcharge sinon
- il peut avoir des redéfinitions sur des méthodes surchargées.
- ce sont deux mécanismes distincts.

```
1  public class A {
2    public void m(String s) { ... } // surcharge
3    public void m(double d) { ... } // surcharge
4  }
5  public class B extends A {
6    public void m(Integer i) { ... } // surcharge
7    public void m(String s) { ... } // rede'finition
8  }
```

# Classes abstraites (1)

classe dont certaines méthodes ne possèdent pas de corps.

- ces méthodes sont dites *abstraites* ;
- utilisation du mot clé `abstract`.

Si une sous-classe, d'une classe abstraite, redéfinit toutes les méthodes abstraite de l'ancêtre, alors elle devient concrète, sinon elle reste abstraite.

on ne peut pas avoir d'instance d'une classe abstraite ! ! !

mais on peut avoir un constructeur appelable par une sous-classe.

# Classes abstraites (2)

```
1  abstract class Forme {
2    public void affiche () {
3      System.out.println ("Je suis " + this.qui_suis_je ());
4    }
5    public abstract String qui_suis_je () ;
6  }
7
8  class Carre extends Forme {
9    public String qui_suis_je () { return ("un carre'") ; }
10 }
11
12 class Cercle extends Forme {
13   public String qui_suis_je () { return ("un cercle") ; }
14 }
15
16 class TestForme {
17   public static void main (String[] args) {
18       Forme c1 = new Cercle();
19       Forme c2 = new Carre();
20       c1.affiche();
21       c2.affiche();
22   }
23 }
```

# Classes abstraites et Interfaces

- classe abstraite avec méthodes concrètes (code)
  factorisation du code
- classe abstraite avec attribut
  - constructeur particulier
    appelable à partir d'un autre constructeur, pas d'un **new**.
- sans méthode ni attribut
  - proche d'une interface
  - mais n'autorise pas l'héritage multiple

implantation différente entre classe abstraite et interface.

exercice : comparer les performances.

# Modifieurs de visibilité

Autorisation d'accès par modifieur de visibilité :

| Modifieur | classe | paquetage | sous-classes | monde |
|---|---|---|---|---|
| public | oui | oui | oui | oui |
| protected | oui | oui | oui | |
| default (rien) | oui | oui | | |
| private | oui | | | |

- une classe a toujours l'accès à tous ses champs ;
- les classes du même paquetage ont accès à tous les champs sauf les privés ;
- une sous-classe d'un autre paquetage n'a pas accès aux champs protégés ;

# Différentes visions

- vue client (accès **public**)
  - constructeurs et méthodes publiques
  - pour les utilisateurs de la classe
- vision héritier (accès **protected**)
  - constructeurs et méthodes protégées (+ publiques)
  - pour les concepteurs de sous-classes
- vision fournisseur (accès **private**)
  - accès public+protégé+privé (dont attributs et méthodes privés)
  - pour les autres concepteurs de la classe

# Modifieurs généraux

- **static** : pour les déclarations de variables et de méthodes de classe
- **final** : pas d'héritage (classe), de redéfinition (méthode), de modification (variable)
- **abstract** : pour une classe ou une méthode non complètement définie
- **public**, **protected**, **private** : modifieurs de visibilité
- autres
  - **strictfp** (Strict floating point)
  - **transient** : ne pas sérialiser un champ
  - **volatile** : mettre à jour une valeur pour de potentiels accès concurrents
  - **synchronized** : poru les méthodes utilisant un moniteur (threads)
  - **native** : pour les méthodes natives appelant du code non Java (C)

# Modifieurs généraux

| **abstract** | | **final** |
|---|---|---|
| doit être sous-classée | **C** | ne peut être sous-classée |
| doit avoir une implantation | **M** | ne peut être raffinée |
| | **V** | ne peut être modifiée |

et toujours **static** :
**M** et **V** : méthode ou variable de classe.
(pas de **this** dans ces méthodes)

- **static final** pour une variable en fait une constante ! ! !
- **final** empêche de raffiner une méthode
- **final** d'une classe empêche le sous-classement.

# Modifieurs de visibilité

- **public** : accessibles par tout objet
- **private** : accessibles dans la classe de définition
- **protected** : accessibles dans les sous-classes et classes du même paquetage.

Remarques

- Il n'y a qu'une seule classe publique dans un fichier .java, celle-ci doit porter le nom du fichier .java ;
- en règle générale, les attributs d'une classe sont déclarés en private, et nécessite de créer des méthodes (get) pour y accéder et (set) pour les modifier.

# Mini-projet : Lecteur de fichiers

Faire un programme :

1. Défini une interface de lecteur de fichiers
2. Plusieurs sous-classe pour différents types de fichiers
3. Utilise une classe abstraite principale pour définir les méthodes qui ne changeront pas d'un fichier à l'autre (lesquelles ?)
4. Implémente une des classes qui affiche le fichier **à l'envers** à l'écran (en termes de lignes)
5. Implémente une des classes qui affiche le fichier de manière **palindromique** (en terme de caractères).
6. Comparateur de fichier type « *diff* »

**45 minutes - Noté**